Andersch, M.; Juurlink, B.; Chi, C. C.

A Benchmark Suite for Evaluating Parallel Programming Models

Introduction and Preliminary Results

Andersch, M.; Juurlink, B.; Chi, C. C. (2011). A Benchmark Suite for Evaluating Parallel Programming Models - Introduction and Preliminary Results. PARS: Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware, 28(1), pp. 7-17. https://hdl.handle.net/20.500.12116/8577

A Benchmark Suite for Evaluating Parallel Programming Models

Introduction and Preliminary Results

Michael Andersch, Ben Juurlink, Chi Ching Chi

Embedded Systems Architectures
TU Berlin
Einsteinufer 17
D-10587 Berlin
andersch@cs.tu-berlin.de
juurlink@cs.tu-berlin.de
cchi@cs.tu-berlin.de

Abstract: The transition to multi-core processors enforces software developers to explicitly exploit thread-level parallelism to increase performance. The associated programmability problem has led to the introduction of a plethora of *parallel programming models* that aim at simplifying software development by raising the abstraction level. Since industry has not settled for a single model, however, multiple significantly different approaches exist. This work presents a benchmark suite which can be used to classify and compare such parallel programming models and, therefore, aids in selecting the appropriate programming model for a given task. After a detailed explanation of the suite's design, preliminary results for two programming models, Pthreads and OmpSs/SMPSs, are presented and analyzed, leading to an outline of further extensions of the suite.

1 Introduction

The move towards multi-core architectures changes the programmer's view of the architecture and introduces yet unresolved programmability issues. Increasing performance now requires the explicit exploitation of thread-level parallelism. The development of parallel programs is generally not a trivial task since an appropriate parallel decomposition of the algorithms needs to be found. Additionally, the programmer usually has to perform architecture-specific optimizations such as thread-to-core mapping or page placement, which could lead to different optimal parallelization strategies for different platforms. Furthermore, the verification and debug processes of such programs introduce additional difficulties caused by the complexity associated with sophisticated threads running in parallel.

All this has led to the introduction of several *programming models* in an attempt to relieve developers partly or completely from such parallel programming issues. These models, however, differ significantly in the provided underlying parallel principles, abstraction

levels, semantics, and syntax.

This work aims at providing some means of comparison by introducing an *evaluation suite* consisting of several applications to examine and classify the features and qualities of shared memory parallel programming models. These applications will not only be used as benchmarks to measure performance levels of programs developed in a particular model, they also allow hands-on examination of the usability and features of that model while actually being ported. Additionally, a first version of this suite is presented along with preliminary results for two currently relevant models. The contributions of this work can be summarized as follows:

- We propose a benchmark suite specifically targeted at the evaluation of performance and usability of parallel programming models rather than parallel machines.
- We focus on modularity and portability for the benchmarks incorporated into the suite.
- We perform a case study, evaluating the novel OmpSs programming model [PBL08], using POSIX threads as a reference for comparison.

This paper is structured as follows. Section 2 encompasses the top-level design decisions made in creating the evaluation suite. In Section 2.1, we define general requirements the suite must fulfill. In Section 2.2, these requirements are utilized to create a preliminary selection of benchmarks which are then presented in a more detailed fashion. A case study using the benchmark suite is presented and analyzed in Section 3. Section 4 discusses related work. Finally, in Section 5, conclusions are drawn and future perspectives are given.

2 Suite Design

On a high level, the benchmark suite must fulfill several critical requirements which can be derived from its objective as a suite to evaluate the programmability and performance of parallel programming models. These criteria will then function as guidelines for the selection of benchmarks and benchmarking practice. In the following section, we identify six such criteria. They assure comparability, fairness and ease of use as well as enabling developers using the suite to gather hands-on experience with the programming model in use.

2.1 General Requirements

1. A broad range of application domains must be covered.
2. Various parallel patterns and characteristics must be covered.
3. Various application sizes must be covered.
4. The suite must include input data sets of varying size.
5. Simplicity, modularity and portability must be ensured.
6. The parallelization approach must be fixed and well documented.

The first and second requirements ensure usability of the suite for a wide range of developers as well as different types of parallel algorithms. The third requirement allows for both a quick mediation of a first impression of a programming model's handling as well as the acquirement of in-depth information. The fourth requirement makes it possible to investigate the sensibility of the parallel programs to changes in input size. The fifth requirement demands easy access to the benchmarks for portability. Last, the sixth requirement ensures comparability over different programming models to make an evaluation possible.

2.2 Benchmark Suite

Table 1: Benchmarks

Name	Type	Application	Domain	Inputs [small/large]	Code Size
c-ray	K	Offline Raytracing	Computer Graphics	18 / 192 objects	500 LOC
md5	K	MD5 Calculation	Cryptography	various	1000 LOC
rgbcmy	K	Color Conversion	Image Processing	3.8 / 30.5 MP	700 LOC
rotate	K	Image Rotation	Image Processing	3.8 / 30.5 MP	1000 LOC
kmeans	K	k-Means Clustering	Artificial Intelligence	various	600 LOC
rot-cc	W	rotate + rgbcmy	Combined Workload	3.8 MP / 30.5 MP	1400 LOC
ray-rot	W	c-ray + rotate	Combined Workload	18 / 192 objects	1300 LOC
h264dec	A	H.264 Decoding	Video Processing	Full HD / QHD video	20000 LOC

The current benchmark suite is presented in Table 1. The **K**, **W**, and **A** identifiers in the second column are a realization of the third requirement, grouping benchmarks into one of three different categories, **K**ernels, **W**orkloads, and **A**pplications.

A kernel consists of (a part of) the extracted core of a real-world application. Kernels are, therefore, comparably small ($<$ 1000 LOC) and exhibit only a single, isolated parallelization pattern.

A workload is either derived by combining several kernels, thereby introducing additional data dependencies and covering more parallelization patterns, or it is a program considered too large to fit in the kernel class, but still only an extracted part of a real application.

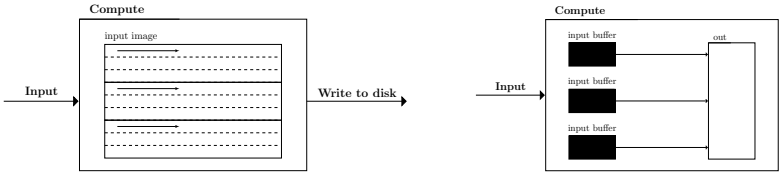
Applications are full-grown software products which are widely used in industry or science and therefore feature the highest number of subsystems, of dependencies and combinations of parallelization patterns.

Following is, in the order depicted in Table 1, a short description of each benchmark. It should be kept in mind this is a preliminary selection which shows the current and not the final state of the suite and is subject to change and extension.

2.2.1 Kernels

c-ray is a simple, brute-force ray tracer [Tsi10]. It is small (ca. 500 LOC for the sequential version) and renders an image in PPM binary format from a simple scene description file. Despite its simplicity, c-ray is a very compute-intensive benchmark, featuring a high computation-to-communication ratio. The parallelization approach is depicted in Figure 1a.

md5 is a benchmark utilizing a standard implementation of the MD5 hash algorithm [Riv92]



(a) c-ray, rgbcmy, rotate (b) md5

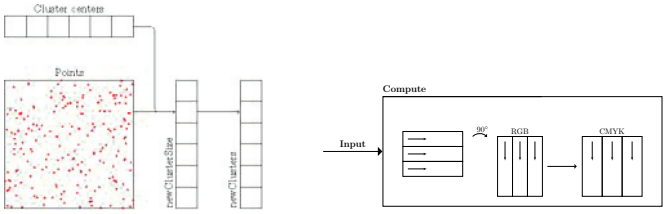
Figure 1: Parallel patterns

to produce hash values. Since there is no exploitable thread-level parallelism in the block cipher construction used in MD5, the benchmark uses multiple input buffers consisting of predefined raw binary data which it processes in parallel. This structure is shown in Figure 1b; parts drawn shaded illustrate exploitable parallelism.

The *rgbcmy* kernel converts an input RGB PPM image to the CMYK color space used for image printing. Parallelism is found in the different pixels (which can be converted independently), visualized in Figure 1a.

rotate is a benchmark which rotates an RGB image in binary representation by 0, 90, 180 or 270 degrees. The parallelization approach can also be visualized by Figure 1a.

The *kmeans* kernel executes the k-Means clustering algorithm [Mac67] used in the domains artificial intelligence and data mining. It is derived from the correspondent benchmark in the NU-MineBench benchmark suite [NOZ+06]. The algorithm is visualized in Figure 2a.



(a) kmeans (b) rot-cc

Figure 2: Parallel patterns

2.2.2 Workloads

As mentioned before, workloads consist of combinations of kernels. The first workload combines the *rotate* and *rgbcmy* kernels and is therefore called *rot-cc* (for rotation + color conversion). The parallelization structure is illustrated in Figure 2b. Interesting cases are those where the rotation changes the image orientation (90 and 270 degrees) since this leads to a strided memory access pattern for the color conversion kernel.

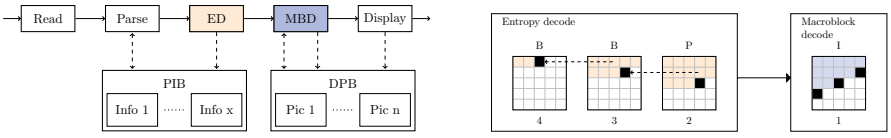
By chaining the *c-ray* and *rotate* kernels, we obtain the *ray-rot* workload. It exhibits additional function-level parallelism and is especially interesting because the two phases

are highly different in characteristics and must, therefore, be load balanced to achieve high performance. *ray-rot* can be visualized as two chained stages of the pattern in Figure 1a.

2.2.3 h264dec

h264dec is an H.264 decoder [CCJ11], derived from FFmpeg, a free, cross-platform H.264 transcoder [FFm10].

For the H.264 decoder benchmark, parallelism is exploited at two levels: function-level and data-level parallelism (DLP). First, in the decoder stages, each stage can be performed in parallel in a pipeline fashion on different frames as shown in Figure 3a. Additionally, DLP is exploited within the entropy (ED) and macroblock decoding (MBD) stages. This is illustrated in Figure 3b. Here, hatched blocks denote data that can be processed in parallel.



(a) Pipeline parallelism

(b) DLP parallelism in ED and MBD stages

Figure 3: Parallel patterns

3 Case study: Pthreads vs. OmpSs

We now present performance results and the documentation of usage experiences comparing two programming models, Pthreads and OmpSs/SMPSs [PBL08]. In Section 3.1, we describe the features of the programming models in comparison. After the experimental setup is presented in Section 3.2, Section 3.3 compares the speedup characteristics of the different benchmarks. From this comparison, we then derive information about the benchmark behavior and gain a first impression on how the two models compare to each other. Section 3.4 completes this insight by documenting the experiences with the usability of each programming model.

3.1 Evaluated Programming Models

The POSIX thread library [IEE04] provides basic threading support for the C programming language. Synchronization is achieved using mutexes to protect critical sections and condition variables to achieve thread synchronization. The threads themselves have to be created, managed (i.e., set to a certain priority or in a detached state) and terminated explicitly. Pthreads thus fully leaves the management of the parallel algorithm to the programmer, enforcing him or her to consider dependencies, synchronization points, and possible race conditions in a direct, exposed way.

OmpSs/**SMPSs** [PBL08] is the **SMP** instance of the *OpenMP SuperScalar model* (OmpSs). It is a novel task-based programming model which consists of a runtime library and a source-to-source compiler. SMPSs requires the programmer to annotate functions as tasks

using *#pragma css task* directives and label every task argument as an *input*, *output*, *inout*, or *reduction* parameter. These keywords declare an argument either read-only, write-only, read-write or as part of a reduction operation. Once such a task is created, it will be added to a runtime data structure, called the *task dependency graph*, which is used by the main thread to perform dependency resolution and the scheduling of tasks on threads. This is similar to the way a superscalar processor dispatches instructions to available execution units. An advantage of SMPSs is that the sequential base code is maintained, allowing profiling and debugging of the sequential code with established tools.

3.2 Experimental Setup

All available results have been obtained during the development of this benchmark suite and are therefore neither specifically optimized nor have been analyzed in detail. Their main objective at this stage is to classify and analyze the early benchmark behavior. Due to this early state of the described benchmarks, results for kmeans and h264dec are not yet included. Our evaluation platform is a 64-core cc-NUMA system with the following features:

Processor	8x Xeon X7560
Architecture	Nehalem EX
Clock Frequency	2.26 GHz
SMT / Turbo mode	Disabled
Main Memory	2 TiB
Aggregate Memory BW	204.8 GiB/s
Compiler	GCC 4.4.3
SMPSs Version	2.3

Each reported result is the average of eight runs. Timing is done using timestamps inside the benchmarks and always excludes the I/O-phases (i.e., loading the input from disk into memory and cleaning up). Additionally, the execution time of all programs has been measured using both a small and a large input data set (see Table 1 for details).

3.3 Preliminary Scaling Results

In this section, the preliminary results for the Pthreads and SMPSs versions of the benchmarks are discussed. The speedup has been obtained by dividing the execution time on one processor by the execution time on n processors *for the same program*, thus normalizing the speedup factor for a single core to one.

The speedup results for up to 64 cores for the Pthreads programming model are shown in Figure 4. The corresponding ones for SMPSs can be found in Figure 5.

The figures show that the behavior varies widely across the applications. For the highest thread count of 64, the Pthreads benchmarks achieve speedup factors ranging from 2.53x to 11.4x for the small and from 10.1x to 31.7x for the large input data sets. For the same

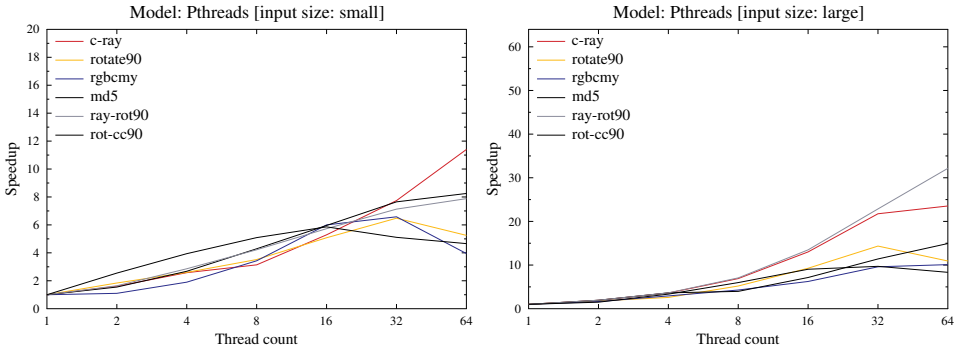


Figure 4: Baseline performance for Pthreads

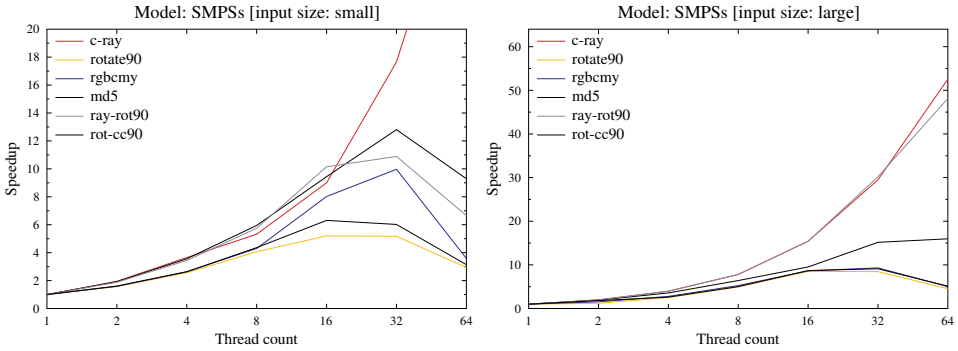


Figure 5: Baseline performance for OmpSs/SMPSs

number of threads, the SMPSs benchmarks achieve speedups between 1.7x and 33.4x for the small and 3.0x and 52.5x for the large input data sets. The differences observed between small and large input sizes are caused by the naturally higher amounts of DLP, leading to a coarsened granularity of the work units and thus diminishing the impact of the threading overhead. The results show that regarding only speedup and not real execution time, for these benchmarks, SMPSs performs on average two times better than Pthreads. For SMPSs, the runtime must be initialized before and shut down after any calls to it are made. This is excluded from the timing, while for Pthreads, thread creation is mostly tightly coupled with the actual execution and is therefore generally included. This is one of the reasons for the higher speedups of SMPSs.

The highest speedups are achieved for benchmarks which include ray tracing. This is expected since c-ray has a high computation-to-communication ratio. The performance of the Pthreads version of c-ray for large inputs saturates, however. In this case, performance increases by only 8% when going from 32 to 64 threads, compared to an average 70% for the other test cases (with small inputs and SMPSs). This is fully reproducable and will be investigated further.

The lowest speedups are achieved by benchmarks which include the rotate kernel (and do not also include ray tracing). The reason for this supposedly is the cc-NUMA evaluation platform. Because there is only a small amount of computation in the rotate kernel, increasing the thread count does not improve performance but instead leads to memory contention, causing a high amount of (coherence) traffic. This is especially the case for the transition from 32 to 64 threads where four additional processor sockets are used for 64 threads, resulting in a lower speedup than for 32 threads.

A deeper analysis of these observations and further machine-specific optimizations are future work.

3.4 Development Experiences

Performance is only a part of the evaluation of a programming model. While higher abstraction levels are targeted by most parallel programming models, it must be questioned whether low-level control can be traded for abstraction. It is a field of active research whether a higher abstraction level can keep the same expressiveness as the usually more complicated, basic counterpart [VPN11]. Therefore, in this section, we want to give exemplary descriptions of notable experiences obtained in the process of parallelizing our benchmarks with the two considered programming models. For each programming model we will discuss the following qualitative and quantitative metrics:

- Code size,
- Code transformations required,
- Expressiveness,
- Tool chain,
- Verification and debugging.

The Pthreads benchmarks are in general larger than their sequential counterparts. This increase is due to the necessity of using specialized threading data structures and multiple function calls. SMPSs has a clear advantage in this respect since the only additions necessary to the code are pragmas to start and end the runtime and declare tasks. This advantage, however, only holds if no manual code transformations have to be applied to expose the parallelism. For Pthreads, such transformations were necessary several times. The programmer must reorganize the code into thread functions and thread inputs must be grouped, if not global, into a single parameter structure. SMPSs, given a sequential program that can be straightforwardly parallelized, only enforces the programmer to decompose the function arguments into parts (because structure member access and pointer dereference is impossible in SMPSs directives) to allow the direct annotation of these parts as inputs or outputs. If the program cannot be straightforwardly parallelized, fundamental changes to the sequential base code might be required.

Regarding the expressiveness of the two models, with Pthreads, any kind of parallelism can be expressed. However, since fine-grained control over all condition and lock variables is required, the program code quickly becomes unintuitive to read. This is especially true for large programs where more parallel threads are interacting. Expressing parallelism

in SMPSs is, on the contrary, much more straightforward. However, our experiences are that several well known parallel programming patterns cannot be conveniently expressed using SMPSs. For example, pipeline parallelism requires buffers between two consecutive pipeline stages to decouple them. With Pthreads, this can be implemented using any kind of dynamic buffers like queues or ringbuffers to communicate between stages. With SMPSs, currently this is not possible.

The use of Pthreads only requires to link the library to include by the C preprocessor and therefore does not impose changes in the toolchain employed. SMPSs, on the contrary, provides a complete runtime system and a specialized compiler, leading to the issue of missing support for several popular compilation parameters.

A well-known problem when developing parallel programs is debugging. It is difficult to debug programs where several things happen at the same time. This applies to Pthreads as well as SMPSs. Debugging an SMPSs program, however, is significantly more difficult for three reasons: First, the source-to-source compiler of the SMPSs programming model makes debugging with a conventional debugger difficult since the code which is debugged is not the code that was actually written. Second, in the SMPSs programming model the programmer has only an abstracted view of the underlying task execution system. In the case of program misbehavior, the raised abstraction level could turn into a serious issue since a clear understanding of the program behavior is required to efficiently locate and eliminate bugs. Third, established tools like gdb lack support for SMPSs-specific language primitives and can therefore only be used to debug SMPSs programs in a naive way. Tools to help debugging SMPSs programs are currently being developed [CRA10].

4 Related Work

Benchmark suites have been developed previously, including several proprietary, domain-specific products [Ber94, Sta]. They are, however, mainly non-compliant with the portability concept our work chooses to follow.

PARSEC [Bie11, BL09] is a recent benchmark suite consisting of 12 programs. The target platforms of PARSEC originally are chip multiprocessors, however, the programs included in the suite are not inherently limited to this usage scenario. The stated goal of PARSEC is to discover new trends in the research and development of parallel machines, algorithms and applications. Featured for all benchmarks are variants for Pthreads, OpenMP and Intel Thread Building Blocks. However, PARSECs set goal is also to provide a fix framework for benchmark execution, input data size control and installation. This complicates the processes of extending the suite quickly or extracting an application out of it for further, isolated use.

Apart from benchmark suites, previous work also includes attempts particularly targeted at the evaluation of parallel programming models.

Podobas et al. [PBF10] performed an evaluation of three task-based parallel programming models, OpenMP, Cilk++ and Wool. They focus on leveraging the performance characteristics of these parallel programming models, studying in detail the cost of creating, spawning and joining tasks as well as overall performance. The results are limited to only

three programming models, only kernel-type programs and only performance characteristics. Moreover, the work excludes the extension to new programming models and therefore is, in contrast to our work, not portable.

Ravela [Rav10] presents an evaluation of Intel TBB, Pthreads, OpenMP and Cilk++, containing results for both achieved performance and the time required to develop the respective versions of the benchmarks. All used benchmarks are, however, taken from the domain of high performance computing, resulting in limited relevance for different application domains.

5 Conclusions and Future Work

In this paper, we presented a benchmark suite to evaluate the programmability and performance of emerging parallel programming models. To achieve this, we focused on a structured, portability-focused, fixed-parallelism approach. We analyzed the intended usage of the suite, thereby compiling a set of requirements which must be met by a benchmark suite aimed at evaluating parallel programming models. We presented and described an early collection of such benchmarks, covering a wide range of application domains, and used them in a case study, comparing an established with an emerging programming model.

The preliminary experimental results obtained in this process have shown a wide range of characteristics for the chosen benchmark set, especially giving insights about the behavioral properties of those benchmarks and producing valuable information on the scaling and speedup characteristics of the two analyzed programming models. This study must also be extended to additional types of parallel machines, for example heterogeneous architectures or large chip multiprocessors. Such an investigation could also include a detailed analysis of the statistical features of each benchmark, resulting in concrete measures for properties such as bandwidth usage, arithmetic complexity or memory size requirements.

Naturally, our benchmark suite will be subject to extension. As mentioned in Section 4, porting suitable, existing open-source benchmarks from other collections to this suite is ongoing work. Furthermore, we seek to extend the suite with additional, industry-relevant applications in order to gain key insights on how modern programming models fare when used in large, real-world applications. Benchmarks currently being considered are dedup and bodytrack from PARSEC. Aside from adding more benchmarks to the suite, another goal is to evaluate more programming models to gather more experiences in using the suite. Evaluating a larger number of programming models is naturally an advantage because it will provide more references to compare to when evaluating new programming models.

Acknowledgment

This research has been supported in part by the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement n° 248647 [Enc], and the Future SOC Lab of Hasso-Plattner-Institute Potsdam [Has].

References

[Ber94] Michael Berry. Public International Benchmarks for Parallel Computers: PARKBENCH Committee: Report-1. *Sci. Program.*, 1994.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[BL09] Christian Bienia and Kai Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proc. 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[CCJ11] Chi Ching Chi and Ben Juurlink. A QHD-Capable Parallel H.264 Decoder. In *Proceedings of the 25th Int. Conf. on Supercomputing*, 2011.

[CRA10] Paul Carpenter, Alex Ramirez, and Eduard Ayguade. Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs. In *Euro-Par 2010 - Parallel Processing*. 2010.

[Enc] Encore Project. ENabling technologies for a future many-CORE.

[FFm10] FFmpeg group. 2010.

[Has] Hasso-Plattner-Institut Potsdam. Future SOC Lab. http://www.hpi.uni-potsdam.de/forschung/future_soc_lab.html.

[IEE04] IEEE Opengroup. Portable Operating System Interface, 2004.

[Mac67] J. B. MacQueen. Some Methods for Classification and Analysis of MultiVariate Observations. In *Proc. of the fifth Berkeley Symp. on Mathematical Statistics and Probability*, 1967.

[NOZ+06] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proc. Int. Symp. on Workload Characterization (IISWC)*, 2006.

[PBF10] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A Comparison of some recent Task-based Parallel Programming Models. In *Third Workshop on Programmability Issues for Multi-Core Computers*, 2010.

[PBL08] J.M. Perez, R.M. Badia, and J. Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. In *Cluster Computing, 2008 IEEE International Conference on*, 2008.

[Rav10] Srikar C. Ravela. Comparison of Shared memory based parallel programming models. Master's thesis, Biekinge Institute of Technology, 2010.

[Riv92] Ronald Rivest. The MD5 Message-Digest Algorithm, 1992.

[Sta] Standard Performance Evaluation Corporation. SPEC Benchmark Suite. http://www.spec.org/index.html.

[Tsi10] John Tsiombikas. 2010.

[VPN11] Hans Vandierendonck, Polyvios Pratikakis, and Dimitrios Nikolopoulos. Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. to be published, 2011.